Improving Size-Change Analysis in Offline Partial Evaluation*

Michael Leuschel[1], Salvador Tamarit[2], and Germán Vidal[2]

[1] Institut für Informatik, Universität Düsseldorf, D-40225, Düsseldorf, Germany
`leuschel@cs.uni-duesseldorf.de`
[2] DSIC, Technical University of Valencia, E-46022, Valencia, Spain
`{stamarit,gvidal}@dsic.upv.es`

Abstract. Some recent approaches for scalable offline partial evaluation of logic programs include a size-change analysis for ensuring both so called local and global termination. In this work—inspired by experimental evaluation—we introduce several improvements that may increase the accuracy of the analysis and, thus, the quality of the associated specialized programs. We aim to achieve this while maintaining the same complexity and scalability of the recent works.

1 Introduction

Partial evaluation [4] is a well-known technique for program specialization. In this work, we consider the so called *offline* approach, which consists of two clearly separated phases: binding-time analysis and proper specialization. Basically, the binding-time analysis should annotate the source code in order to drive the specialization process. Roughly speaking,

– every atom is annotated as either unfold (the atom can be unfolded) or memo (the atom should not be unfolded), and
– every predicate's argument is classified as either static (definitely known at specialization time) or dynamic (possibly unknown at specialization time).

We say that the annotations are *safe* if static arguments are actually ground at specialization time and the termination of the specialization is ensured. Termination issues are usually classified into local and global termination:

– local termination ensures that no atom is infinitely unfolded;
– global termination guarantees that only finitely many atoms are specialized (i.e., that we do not create infinite specializations of the same predicate).

The main component of a binding-time analysis is a termination analysis that allows us to guarantee both local and global termination of the specialization

* This work has been partially supported by the EU (FEDER) and the Spanish MEC/MICINN under grants TIN2005-09207-C03-02, TIN2008-06622-C03-02, and *Acción Integrada* HA2006-0008

process. In [8], a *strong* termination analysis—based on the so called size-change termination principle [5]—for logic programs is introduced. Strong termination means termination w.r.t. all selection rules. Although this is a rather strong condition, it allows us to design much faster binding-time analysis (see [6]).

In this paper, we identify several weaknesses of the original size-change analysis of [8] and present different proposals that improve the accuracy of the specialization process.

2 Size-Change Termination Analysis

In this section, we informally present the basis of the quasi-termination analysis for logic programs of [8].

We say that a query Q is *strongly terminating* w.r.t. a program P if every SLD derivation for Q with P is finite. We denote by $calls_P^{\mathcal{R}}(Q_0)$ the set of calls in the computations of a goal Q_0 within a logic program P and a computation rule $\mathcal{R}$. The query Q is *strongly quasi-terminating* if, for every computation rule $\mathcal{R}$, the set $call_P^{\mathcal{R}}(Q)$ contains finitely many nonvariant atoms. A program P is strongly (quasi-)terminating w.r.t. a set of queries $\mathcal{Q}$ if every $Q \in \mathcal{Q}$ is strongly (quasi-)terminating w.r.t. P. For conciseness, in the remainder of this paper, we write "(quasi-)termination" to refer to "strong (quasi-)termination."

Size-change analysis is based on constructing graphs that represent the decrease of the arguments of a predicate from one call to another. For this purpose, some ordering on terms is required.

Definition 1 (reduction pair). *We say that $(\succsim, \succ)$ is a reduction pair if $\succsim$ is a quasi-order and $\succ$ is a well-founded order where both $\succsim$ and $\succ$ are closed under substitutions and compatible (i.e., $\succsim \circ \succ \subseteq \succ$ and $\succ \circ \succsim \subseteq \succ$ but $\succsim \subseteq \succ$ is not necessary).*

In logic programming, however, termination analyses usually rely on the use of *norms* which measure the size of terms. In [8], reduction orders $(\succsim, \succ)$ *induced* from symbolic norms $||\cdot||$ are used:

Definition 2 (symbolic norm [3,7]). *Given a term t,*

$$||t|| = \begin{cases} m + \sum_{i=1}^{n} k_i ||t_i|| & \text{if } t = f(t_1, \ldots, t_n),\ n \geqslant 0 \\ t & \text{if } t \text{ is a variable} \end{cases}$$

where m and $k_1, \ldots, k_n$ are non-negative integer constants depending only on f/n. Note that we associate a variable over integers with each logical variable (we use the same name for both since the meaning is clear from the context).

The introduction of variables in the range of the norm provides a simple mechanism to express dependencies between the sizes of terms.

The associated induced orders $(\succsim, \succ)$ are defined as follows: $t_1 \succ t_2$ (respec. $t_1 \succsim t_2$) if $||t_1\sigma|| > ||t_2\sigma||$ (respec. $||t_1\sigma|| \geqslant ||t_2\sigma||$) for all substitution σ that makes $||t_1\sigma||$ and $||t_2\sigma||$ ground (e.g., an integer constant). Two popular instances

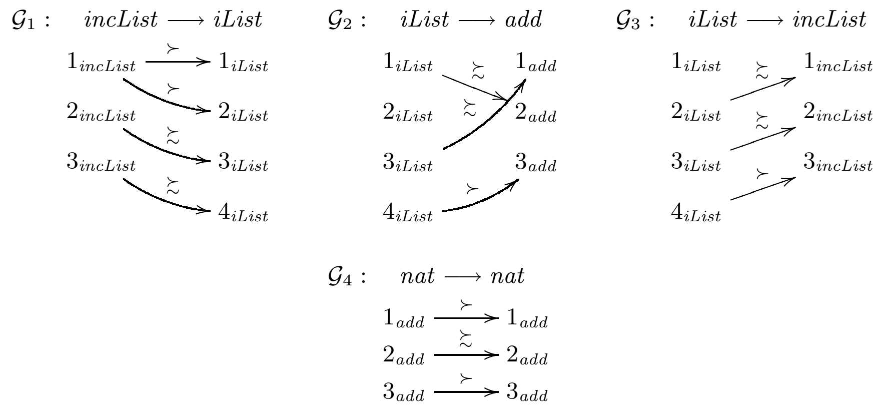

Fig. 1. Size-change graphs for *incList*

of symbolic norms are the symbolic *term-size* norm $||\cdot||_{ts}$ (which sums the arities of the term symbols) and the symbolic *list-length norm* $||\cdot||_{ll}$ (which counts the number of elements of a list), e.g.,

$$f(X,Y,a) \succ_{ts} f(X,a,b) \text{ since } ||f(X,Y,a)||_{ts} = X+Y+3 > X+3 = ||f(X,a,b)||_{ts}$$
$$[X|R] \succsim_{ll} [s(X)|R] \text{ since } ||[X|R]||_{ll} = R+1 \geqslant R+1 = ||[s(x)|R]||_{ll}$$

Now, we produce a *size-change graph* $\mathcal{G}$ for every pair (H, B_i) of every clause $H \leftarrow B_1, \ldots, B_n$ of the program, with edges between the arguments of H and B_i when the size of the corresponding terms decrease w.r.t. a given reduction pair $(\succsim, \succ)$.

Example 1. Consider the following simple program:

(c_1) $incList([\,], \_, [\,])$.
(c_2) $incList([X|R], I, L) \leftarrow iList(X, R, I, L)$.
(c_3) $iList(X, R, I, [XI|RI]) \leftarrow add(I, X, XI), incList(R, I, RI)$.
(c_4) $add(0, Y, Y)$.
(c_5) $add(s(X), Y, s(Z)) \leftarrow add(X, Y, Z)$.

Let $(\succsim, \succ)$ be the reduction pair induced by the symbolic term-size norm $||\cdot||_{ts}$. Here, we have four size-change graphs, depicted in Fig. 1, which are associated to clauses c_2 (graph $\mathcal{G}_1$), c_3 (graphs $\mathcal{G}_2$ and $\mathcal{G}_3$) and c_5 (graph $\mathcal{G}_4$).

In order to identify the program *loops*, we should compute roughly a transitive closure of the size-change graphs by composing them in all possible ways. Basically, given two size-change graphs:

$$\mathcal{G} = (\{1_p, \ldots, n_p\}, \{1_q, \ldots, m_q\}, E_1) \qquad \mathcal{H} = (\{1_q, \ldots, m_q\}, \{1_r, \ldots, l_r\}, E_2)$$

w.r.t. the same reduction pair $(\succsim, \succ)$, their concatenation is defined by

$$\mathcal{G} \bullet \mathcal{H} = (\{1_p, \ldots, n_p\}, \{1_r, \ldots, l_r\}, E)$$

where E contains an edge from i_p to k_r iff E_1 contains an edge from i_p to some j_q and E_2 contains an edge from j_q to k_r. Furthermore, if some of the edges are labeled with $\succ$, then so is the edge in E; otherwise, it is labeled with $\succsim$.

In particular, according to [5], we only need to consider the *idempotent* size-change graphs $\mathcal{G}$ with $\mathcal{G} \bullet \mathcal{G} = \mathcal{G}$ for analyzing the termination of the program.

Example 2. For the program of Example 1, we compute the following idempotent size-change graphs:

$$
\begin{array}{ccc}
\mathit{incList} \longrightarrow \mathit{incList} & \quad \mathit{iList} \longrightarrow \mathit{iList} & \quad \mathit{add} \longrightarrow \mathit{add} \\
\begin{array}{l}
1_{incList} \xrightarrow{\succ_{ts}} 1_{incList} \\
2_{incList} \xrightarrow{\succsim_{ts}} 2_{incList} \\
3_{incList} \xrightarrow{\succ_{ts}} 3_{incList} \\
\\
\end{array}
&
\begin{array}{l}
1_{iList} \xrightarrow{\succ_{ts}} 1_{iList} \\
2_{iList} \xrightarrow{\succ_{ts}} 2_{iList} \\
3_{iList} \xrightarrow{\succsim_{ts}} 3_{iList} \\
4_{iList} \xrightarrow{\succ_{ts}} 4_{iList}
\end{array}
&
\begin{array}{l}
1_{add} \xrightarrow{\succ_{ts}} 1_{add} \\
2_{add} \xrightarrow{\succsim_{ts}} 2_{add} \\
3_{add} \xrightarrow{\succ_{ts}} 3_{add} \\
\\
\end{array}
\end{array}
$$

that represent how the size of the arguments of the three potentially looping predicates changes from one call to another.

Once the idempotent size-change graphs of a program have been computed, the following results hold:[3]

Termination: An atom A is (strongly) terminating if every idempotent size-change graph for p/n contains at least one edge $i_p \xrightarrow{\succ} i_p$ such that, for every computation rule $\mathcal{R}$ and atom $p(t_1, \ldots, t_n) \in \mathit{calls}_P^{\mathcal{R}}(A)$, the argument t_i is instantiated enough w.r.t. the considered symbolic norm.
Clearly, the set $\mathit{calls}_P^{\mathcal{R}}(A)$ is often infinite. Therefore, we usually consider an approximation based on a *division* that classifies every predicate's argument as either static or dynamic and check that the i-th argument of p is classified as static (rather than checking that t_i is instantiated enough in all possible calls from A).
For instance, given a division that classifies the arguments of *add* as follows:

$$\mathit{add} \;\mapsto\; (\mathsf{static}, \mathsf{dynamic}, \mathsf{dynamic})$$

and according to the idempotent size-change graphs of Example 2, we have that all calls to *add* terminate since there is an edge $1_{add} \xrightarrow{\succ} 1_{add}$ in the idempotent size-change graph and the first argument of *add* is classified as static.

Quasi-termination: An atom A is (strongly) quasi-terminating if it is either terminating or every idempotent size-change graph for p/n contains, for all i_p $(i = 1, \ldots, n)$ an edge $j_p \xrightarrow{R} i_p$ for some j_p, with $R \in \{\succ, \succsim\}$ (i.e., all arguments are bounded by the value of some argument in a previous call). Furthermore, the considered norms must be *bounded* (see Definition 3 below).

[3] A term t is *instantiated enough* [3,7] w.r.t. a symbolic norm $||\cdot||$ if $||t||$ is an integer constant.

For instance, according to the idempotent size-change graphs of Example 2, an atom $add(X, Y, Z)$ is quasi-terminating since there is an input edge to every argument.

In [8], the termination condition is used for ensuring the local termination of partial evaluation, while the quasi-termination condition is used for ensuring its global termination. Basically,

– we reclassify as unfold those atoms which are terminating w.r.t. a given division (and with memo otherwise) and
– we mark with dynamic the argument of an atom if there is no input edge to this argument in some idempotent size-change graph, i.e., if the atom is not quasi-terminating.

Example 3. Given the idempotent size-change graphs of Example 2 and a division that classifies the predicates' arguments as follows:

$$\begin{array}{l} incList \;\mapsto\; (\mathsf{dynamic}, \mathsf{static}, \mathsf{dynamic}) \\ iList \;\mapsto\; (\mathsf{dynamic}, \mathsf{dynamic}, \mathsf{static}, \mathsf{dynamic}) \\ add \;\mapsto\; (\mathsf{static}, \mathsf{dynamic}, \mathsf{dynamic}) \end{array}$$

we have that

– *incList* and *iList* are marked with memo while *add* is marked with unfold, and
– no argument should be re-classified as dynamic.

3 Improving Size-Change Analysis

In this section, we introduce several extensions of the size-change analysis that may improve the accuracy of the specialization process by taking into account some basic properties of partial evaluation.

3.1 Non-Bounded Norms for Global Termination

Let us recall the notion of *bounded* norm required in [8] for ensuring quasi-termination:

Definition 3 (bounded norm). *We say that a symbolic norm $||\cdot||$ is bounded if the set $\{s \mid ||t|| \geqslant ||s||\}$ contains a finite number of nonvariant terms for any term t.*

Roughly speaking, a symbolic norm is bounded if, for every term t, there exist only finitely many nonvariant terms whose weights are lesser than or equal to that of t w.r.t. the symbolic norm $||\cdot||$.

Unfortunately, many symbolic norms are not bounded; e.g., the symbolic list-length norm is not bounded since, given the term $p([a])$, we have an infinite

set $\{p([a]), p([f(a)]), p([f(f(a))]), \ldots\}$ of non-variant terms such that $||[a]||_{ll} = ||[f(a)]||_{ll} = ||[f(f(a))]||_{ll} = \ldots = 1$.

In the context of partial evaluation, however, symbolic norms need not be bounded if the *problematic* parts of the terms are generalized at the global level. For instance, we can safely use the symbolic list-length norm as long as the list elements are replaced by fresh variables in the global level. This idea, already sketched in [6], is formalized by means of the *most general generalization* operator:

Definition 4 (*mgg*). *Let $||\cdot||$ be a symbolic norm. Given a term t, we denote by $mgg^{||\cdot||}(t)$ the most general generalization of t such that $||t|| = ||mgg^{||\cdot||}(t)||$. We also let $mgg^{||\cdot||}(p(t_1, \ldots, t_n)) = p(mgg^{||\cdot||}(t_1), \ldots, mgg^{||\cdot||}(t_n))$.*

For instance, given the term $t = [s(N), b]$, we have $mgg^{||\cdot||_{ll}}(t) = [X, Y]$ but $mgg^{||\cdot||_{ts}}(t) = [s(N), b]$.

Moreover, the quasi-termination result in [8] also requires that all calls encountered during partial evaluation should be *linear* w.r.t. the dynamic variables (i.e., no variable marked as dynamic could appear more than once in a call). However, this is not a real problem in the context of partial evaluation since all dynamic parts of terms are replaced by fresh variables in the global level anyway.

Therefore, one can ensure the global termination of partial evaluation when using arbitrary symbolic norms in the size-change analysis as long as

- dynamic parts of arguments are replaced by fresh variables in the global level (this is already done by current offline partial evaluators) and
- an atom A is replaced by $mgg^{||\cdot||}(A)$ in the global level, where $||\cdot||$ is the symbolic norm used in the size-change analysis.

3.2 Maximizing "Unfold" Annotations

The original approach of [8] does not take into account that different idempotent size-change graphs may represent a single loop. For instance, the idempotent size-change graphs for both *incList* and *iList* actually represent the same program loop. Therefore, it would be safe to annotate only one of these predicates with "memo" and the other one with "unfold".

In order to avoid unnecessary memo annotations, one can slightly extend the original annotation procedure as follows:

- First, every size-change graph is labeled with a unique identifier (e.g., $\mathcal{G}_1$, $\mathcal{G}_2$, ..., as in Fig. 1).
- Then, the concatenation of graphs is performed as before, but now every concatenation keeps a set with the identifiers of the graphs involved in the concatenation. We note that the set of identifiers is not taken into account during the concatenation process, i.e., two size-change graphs that only differ in the associated set of identifiers are considered equal (therefore, the complexity of the concatenation process, the most expensive part of the analysis, remains the same).

  For instance, the labeled idempotent size-change graphs of Example 2 would now be as depicted in Fig. 2.

$$
\begin{array}{ccc}
\{\mathcal{G}_1, \mathcal{G}_3\} : incList \quad \longrightarrow \quad incList & \{\mathcal{G}_1, \mathcal{G}_3\} : iList \quad \longrightarrow \quad iList & \{\mathcal{G}_4\} : add \quad \longrightarrow \quad add \\
\begin{array}{l}
1_{incList} \xrightarrow{\succ_{ts}} 1_{incList} \\
2_{incList} \xrightarrow{\succsim_{ts}} 2_{incList} \\
3_{incList} \xrightarrow{\succ_{ts}} 3_{incList}
\end{array}
&
\begin{array}{l}
1_{iList} \xrightarrow{\succ_{ts}} 1_{iList} \\
2_{iList} \xrightarrow{\succ_{ts}} 2_{iList} \\
3_{iList} \xrightarrow{\succsim_{ts}} 3_{iList} \\
4_{iList} \xrightarrow{\succ_{ts}} 4_{iList}
\end{array}
&
\begin{array}{l}
1_{add} \xrightarrow{\succ_{ts}} 1_{add} \\
2_{add} \xrightarrow{\succsim_{ts}} 2_{add} \\
3_{add} \xrightarrow{\succ_{ts}} 3_{add}
\end{array}
\end{array}
$$

Fig. 2. Labeled idempotent size-change graphs for *incList*

- The computed idempotent size-change graphs can now be grouped into equivalence classes so that two idempotent size-change graphs belong to the same class if they are labeled with the same set of identifiers.
- Finally, we should only annotate with "memo" one predicate for every equivalence class of idempotent size-change graphs.

For instance, as mentioned in Example 3, both *incList* and *iList* are marked with memo in the original framework. Now, however, only one of them would be marked with memo (and the other one with unfold).

Clearly, there is a degree of freedom when choosing which is the idempotent size-change graph of a given class that should be marked with memo. For this purpose, one can define appropriate heuristics that minimize the number of memo annotations by, e.g., assigning a higher priority to those predicates that belong to more than one class.

3.3 Right-Propagation of Bindings

An advantage of the size-change analysis of [8] is that it is independent of a particular selection rule. As mentioned in the introduction, this property makes the associated binding-time analysis much faster; unfortunately, it is also less accurate.

In some cases, we can improve this situation by assuming some partial knowledge on the evaluation order.[4] For instance, we could first run a left-termination analysis (like, e.g., the one based on the binary unfoldings [2]) or rely on user's annotations that identify some atoms as "completely unfoldable" (note that an annotation unfold only means that the atom can be unfolded *one step*; then the annotations of the predicates in the unfolded goal should be followed).

In this case, we can improve the accuracy of the size-change analysis by using an inter-argument size analysis like that calculated from the convex hull of [1]. For instance, given the program

$$
\begin{array}{l}
p(X) \;\leftarrow\; q(X,Y), p(Y). \\
q(s(0), 0). \\
q(s(X), Y) \;\leftarrow\; q(X,Y).
\end{array}
$$

[4] We thank Maurice Bruynooghe for suggesting this improvement.

the size-change graph associated to $p/1$ originally contains no edge (since we do not know the size relation between X and Y). Now, if we assume that $q/2$ is completely unfoldable, then we can use the output of the convex hull of [1] (using a term-size norm):

$$q(A, B) \;\leftarrow\; \{A > B,\ B = 0,\ A \geq 1\}$$

for propagating some additional constraints to the right of q. In this way, one can easily infer that the size-change graph for $p/1$ should contain an edge $1_p \xrightarrow{\succ} 1_p$.

Let us note that, in principle, the accuracy of the size-change analysis of [8] could not be improved by adding inter-argument size relations to size-change graphs, since inter-argument relations usually require the atoms to be completely unfolded (i.e., they represent relations that hold for success patterns). This assumption is not generally true in the setting of [8] where partial evaluations are possible.

4 Discussion

We have recently undertaken the implementation of a binding-time analysis for the offline partial evaluation of Prolog programs which is based on the size-change analysis of [8]. In this paper, we have introduced several improvements that may allow us to overcome the main weaknesses of [8]. An experimental evaluation will be conducted in order to assess their effectiveness in practice.

References

1. F. Benoy, A. King, and F. Mesnard. Computing convex hulls with a linear solver. *TPLP*, 5(1-2):259–271, 2005.
2. M. Codish and C. Taboch. A Semantic Basis for the Termination Analysis of Logic Programs. *Journal of Logic Programming*, 41(1):103–123, 1999.
3. N. Dershowitz, N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. A General Framework for Automatic Termination Analysis of Logic Programs. *Appl. Algebra Eng. Commun. Comput.*, 12(1/2):117–156, 2001.
4. N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Englewood Cliffs, NJ, 1993.
5. C.S. Lee, N.D. Jones, and A.M. Ben-Amram. The Size-Change Principle for Program Termination. *SIGPLAN Notices (Proc. of POPL'01)*, 28:81–92, 2001.
6. M. Leuschel and G. Vidal. Fast Offline Partial Evaluation of Large Logic Programs. In *Proc. of the 18th Int'l Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2008)*. Technical University of Valencia, 2008. Available from `http://www.dsic.upv.es/~gvidal/german/papers.html`.
7. N. Lindenstrauss and Y. Sagiv. Automatic Termination Analysis of Logic Programs. In *Proc. of Int'l Conf. on Logic Programming (ICLP'97)*, pages 63–77. The MIT Press, 1997.
8. G. Vidal. Quasi-Terminating Logic Programs for Ensuring the Termination of Partial Evaluation. In *Proc. of the ACM SIGPLAN 2007 Workshop on Partial Evaluation and Program Manipulation (PEPM'07)*, pages 51–60. ACM Press, 2007.